mikel / mail

Watch 72 Star 2,565 Fork 606

A Really Ruby Mail Library

1,627 commits 11 branches 51 releases 155 contributors MIT

Branch: master New pull request Find file Clone or download

jeremy Ragel: drop source line numbers from generated parsers since any defi... ... Latest commit 88b1769 18 days ago

gemfiles	rake requires 1.9.3 starting with v11.0.0	7 months ago
lib	Ragel: drop source line numbers from generated parsers since any defi...	18 days ago
patches	Adding patch file	6 years ago
reference	Replace Treetop parser with a Ragel based parser	4 years ago
spec	Ragel: merge machines into parsers	20 days ago
tasks	Ragel: drop source line numbers from generated parsers since any defi...	18 days ago
.gitignore	Modular Ragel definitions and incremental parser builds	27 days ago
.travis.yml	CI: allow jruby-9.0.5.0 failures,, needs work	5 months ago
Appraisals	update gemfiles and exclude jruby	9 months ago
CHANGELOG.rdoc	Fix issue decoding base64 on some rubies	5 months ago
CONTRIBUTING.md	Add advice for iterating on pull requests	2 years ago
Dependencies.txt	Treetop is no longer a dependency	2 years ago
Gemfile	Pin mime-types to < 3.0 on Ruby 1.9.3	6 months ago
MIT-LICENSE	Version bump to 2.6.4 and release	7 months ago
README.md	Merge pull request #962 from pra85/patch-1	7 months ago
ROADMAP	Updating Roadmap	7 years ago
Rakefile	Warn that Appraisal is 1.9+ when it gets loaded on Ruby 1.8	2 years ago
TODO.rdoc	Fixing up TODO and docs	7 years ago
mail.gemspec	mail works with mime-types 3.0	11 months ago
test.rb	Remove ruby-debug from dependencies.	10 months ago

README.md

Mail

build passing

Introduction

Mail is an internet library for Ruby that is designed to handle emails generation, parsing and sending in a simple, rubyesque manner.

The purpose of this library is to provide a single point of access to handle all email functions, including sending and receiving emails. All network type actions are done through proxy methods to Net::SMTP, Net::POP3 etc.

Built from my experience with TMail, it is designed to be a pure ruby implementation that makes generating, sending and parsing emails a no brainer.

It is also designed from the ground up to work with the more modern versions of Ruby. This is because Ruby > 1.9 handles text encodings much more wonderfully than Ruby 1.8.x and so these features have been taken full advantage of in this library allowing Mail to handle a lot more messages more cleanly than TMail. Mail does run on Ruby 1.8.x... it's just not as fun to code.

Finally, Mail has been designed with a very simple object oriented system that really opens up the email messages you are parsing, if you know what you are doing, you can fiddle with every last bit of your email directly.

Donations

Mail has been downloaded millions of times, by people around the world, in fact, it represents more than 1% of *all* gems downloaded.

It is (like all open source software) a labour of love and something I am doing with my own free time. If you would like to say thanks, please feel free to make a donation and feel free to send me a nice email :)

Compatibility

Every Mail commit is tested by Travis on the following platforms

- ruby-1.8.7 [i686]
- ruby-1.9.2 [x86_64]
- ruby-1.9.3 [x86_64]
- ruby-2.0.0 [x86_64]
- ruby-2.1.2 [x86_64]
- ruby-head [x86_64]
- jruby [x86_64]
- jruby-head [x86_64]
- rbx-2 [x86_64]

Testing a specific mime type (needed for 1.8.7 for example) can be done manually with:

```
BUNDLE_GEMFILE=gemfiles/mime_types_1.16.gemfile (bundle check      || bundle)  && rake
```

Discussion

If you want to discuss mail with like minded individuals, please subscribe to the Google Group.

Current Capabilities of Mail

- RFC2822 Support, Reading and Writing
- RFC2045-2049 Support for multipart emails
- Support for creating multipart alternate emails
- Support for reading multipart/report emails & getting details from such
- Support for multibyte emails - needs quite a lot of work and testing
- Wrappers for File, Net/POP3, Net/SMTP

- Auto encoding of non US-ASCII header fields
- Auto encoding of non US-ASCII bodies

Mail is RFC2822 compliant now, that is, it can parse and generate valid US-ASCII emails. There are a few obsoleted syntax emails that it will have problems with, but it also is quite robust, meaning, if it finds something it doesn't understand it will not crash, instead, it will skip the problem and keep parsing. In the case of a header it doesn't understand, it will initialise the header as an optional unstructured field and continue parsing.

This means Mail won't (ever) crunch your data (I think).

You can also create MIME emails. There are helper methods for making a multipart/alternate email for text/plain and text/html (the most common pair) and you can manually create any other type of MIME email.

Roadmap

Next TODO:

- Improve MIME support for character sets in headers, currently works, mostly, needs refinement.

Testing Policy

Basically... we do BDD on Mail. No method gets written in Mail without a corresponding or covering spec. We expect as a minimum 100% coverage measured by RCov. While this is not perfect by any measure, it is pretty good. Additionally, all functional tests from TMail are to be passing before the gem gets released.

It also means you can be sure Mail will behave correctly.

Note: If you care about core extensions (aka "monkey-patching"), please read the Core Extensions section near the end of this README.

API Policy

No API removals within a single point release. All removals to be deprecated with warnings for at least one MINOR point release before removal.

Also, all private or protected methods to be declared as such - though this is still I/P.

Installation

Installation is fairly simple, I host mail on rubygems, so you can just do:

```
# gem install mail
```

Encodings

If you didn't know, handling encodings in Emails is not as straight forward as you would hope.

I have tried to simplify it some:

1. All objects that can render into an email, have an `#encoded` method. Encoded will return the object as a complete string ready to send in the mail system, that is, it will include the header field and value and CRLF at the end and wrapped as needed.

2. All objects that can render into an email, have a `#decoded` method. Decoded will return the object's "value" only as a string. This means it will not include the header fields (like 'To:' or 'Subject:').

3. By default, calling `#to_s` on a container object will call its encoded method, while `#to_s` on a field object will call its decoded method. So calling `#to_s` on a Mail object will return the mail, all encoded ready to send, while calling `#to_s` on the From field or the body will return the decoded value of the object. The header object of Mail is considered a container. If you are in doubt, call `#encoded`, or `#decoded` explicitly, this is safer if you are not sure.

4. Structured fields that have parameter values that can be encoded (e.g. ContentType) will provide decoded parameter values when you call the parameter names as methods against the object.

5. Structured fields that have parameter values that can be encoded (e.g. ContentType) will provide encoded parameter values when you call the parameter names through the `object.parameters['']` method call.

Contributing

Please do! Contributing is easy in Mail.Please read the CONTRIBUTING.md document for more info

Usage

All major mail functions should be able to happen from the Mail module. So, you should be able to ju `require 'mail'` to get started.

Making an email

```
mail = Mail.new do
  from    'mikel@test.lindsaar.net'
  to      'you@test.lindsaar.net'
  subject 'This is a test email'
  body    File.read('body.txt')
end

mail.to_s #=> "From: mikel@test.lindsaar.net\r\nTo: you@...
```

Making an email, have it your way:

```
mail = Mail.new do
  body File.read('body.txt')
end

mail['from'] = 'mikel@test.lindsaar.net'
mail[:to]    = 'you@test.lindsaar.net'
mail.subject = 'This is a test email'

mail.header['X-Custom-Header'] = 'custom value'

mail.to_s #=> "From: mikel@test.lindsaar.net\r\nTo: you@...
```

Don't Worry About Message IDs:

```
mail = Mail.new do
  to   'you@test.lindsaar.net'
  body 'Some simple body'
end

mail.to_s =~ /Message\-ID: <[\d\w_]+@.+.mail/ #=> 27
```

Mail will automatically add a Message-ID field if it is missing and give it a unique, random Message-ID along the lines of:

```
<4a7ff76d7016_13a81ab802e1@local.host.mail>
```

Or do worry about Message-IDs:

```
mail = Mail.new do
  to         'you@test.lindsaar.net'
  message_id '<ThisIsMyMessageId@some.domain.com>'
  body       'Some simple body'
end

mail.to_s =~ /Message\-ID: <ThisIsMyMessageId@some.domain.com>/ #=> 27
```

Mail will take the message_id you assign to it trusting that you know what you are doing.

Sending an email:

Mail defaults to sending via SMTP to local host port 25If you have a sendmail or postfix daemon running on this port, sending email is as easy as:

```
Mail.deliver do
  from     'me@test.lindsaar.net'
  to       'you@test.lindsaar.net'
  subject  'Here is the image you wanted'
  body     File.read('body.txt')
  add_file '/full/path/to/somefile.png'
end
```

or

```
mail = Mail.new do
  from     'me@test.lindsaar.net'
  to       'you@test.lindsaar.net'
  subject  'Here is the image you wanted'
  body     File.read('body.txt')
  add_file :filename => 'somefile.png', :content => File.read('/somefile.png')
end

mail.deliver!
```

Sending via sendmail can be done like so:

```
mail = Mail.new do
  from     'me@test.lindsaar.net'
  to       'you@test.lindsaar.net'
  subject  'Here is the image you wanted'
  body     File.read('body.txt')
  add_file :filename => 'somefile.png', :content => File.read('/somefile.png')
end

mail.delivery_method :sendmail

mail.deliver
```

Sending via smtp (for example tomailcatcher)

```
Mail.defaults do
  delivery_method :smtp, address: "localhost", port: 1025
end
```

Exim requires its own delivery manager, and can be used like so:

```
mail.delivery_method  :exim, :location => "/usr/bin/exim "

mail.deliver
```

Getting emails from a pop server:

You can configure Mail to receive email using `retriever_method` within `Mail.defaults` :

```
Mail.defaults do
  retriever_method :pop3, :address    => "pop.gmail.com ",
                          :port       => 995,
                          :user_name  => '<username> ',
                          :password   => '<password> ',
                          :enable_ssl => true
end
```

You can access incoming email in a number of ways.

The most recent email:

```
Mail.all     #=> Returns an array of all emails
Mail.first   #=> Returns the first unread email
Mail.last    #=> Returns the last unread email
```

The first 10 emails sorted by date in ascending order:

```
emails = Mail.find( :what => :first, :count => 10, :order => :asc )
emails.length  #=> 10
```

Or even all emails:

```
emails = Mail.all
emails.length  #=> LOTS!
```

Reading an Email

```
mail = Mail.read( '/path/to/message.eml ')

mail.envelope_from     #=> 'mikel@test.lindsaar.net'
mail.from.addresses    #=> ['mikel@test.lindsaar.net', 'ada@test.lindsaar.net']
mail.sender.address    #=> 'mikel@test.lindsaar.net'
mail.to                #=> 'bob@test.lindsaar.net'
mail.cc                #=> 'sam@test.lindsaar.net'
mail.subject           #=> "This is the subject"
mail.date.to_s         #=> '21 Nov 1997 09:55:06 -0600'
mail.message_id        #=> '<4D6AA7EB.6490534@xxx.xxx>'
mail.body.decoded      #=> 'This is the body of the email...
```

Many more methods available.

Reading a Multipart Email

```
mail = Mail.read( 'multipart_email ')

mail.multipart?            #=> true
mail.parts.length          #=> 2
mail.body.preamble         #=> "Text before the first part"
mail.body.epilogue         #=> "Text after the last part"
```

```
mail.parts.map { | p| p.content_type }    #=> ['text/plain', 'application/pdf']
mail.parts.map { | p| p.class }           #=> [Mail::Message, Mail::Message]
mail.parts[ 0].content_type_parameters    #=> {'charset' => 'ISO-8859-1'}
mail.parts[ 1].content_type_parameters    #=> {'name' => 'my.pdf'}
```

Mail generates a tree of parts.Each message has many or no parts.Each part is another message which can have many or no parts.

A message will only have parts if it is a multipart/mixed or multipart/related content type and has a boundary defined.

Testing and extracting attachments

```
mail.attachments.each  do | attachment  |
  # Attachments is an AttachmentsList object containing a
  # number of Part objects
  if (attachment.content_type.start_with?(   'image/ '))
    # extracting images for example...
    filename  = attachment.filename
    begin
      File .open(images_dir  + filename,  "w+b", 0644 ) {|f| f.write attachment.body.decoded}
    rescue  => e
      puts  "Unable to save data for   #{filename } because  #{e.message }"
    end
  end
end
```

Writing and sending a multipart/alternative (html and text) email

Mail makes some basic assumptions and makes doing the common thing as simple as possible.... (asking a lot from a mail library)

```
mail  = Mail .deliver  do
  to       'nicolas@test.lindsaar.net.au  '
  from     'Mikel Lindsaar <mikel@test.lindsaar.net.au>   '
  subject  'First multipart email sent with Mail   '

  text_part  do
    body  'This is plain text  '
  end

  html_part  do
    content_type  'text/html; charset=UTF-8  '
    body  '<h1>This is HTML</h1>  '
  end
end
```

Mail then delivers the email at the end of the block and returns the resulting Mail::Message object, which you can then inspect if you so desire...

```
puts mail.to_s #=>

To: nicolas@test.lindsaar.net.au
From: Mikel Lindsaar <mikel@test.lindsaar.net.au>
Subject: First multipart email sent with Mail
Content-Type: multipart/alternative;
  boundary=--==_mimepart_4a914f0c911be_6f0f1ab8026659
Message-ID: <4a914f12ac7e_6f0f1ab80267d1@baci.local.mail>
Date: Mon, 24 Aug 2009 00:15:46 +1000
Mime-Version: 1.0
Content-Transfer-Encoding: 7bit


-----==_mimepart_4a914f0c911be_6f0f1ab8026659
Content-ID: <4a914f12c8c4_6f0f1ab80268d6@baci.local.mail>
```

```
Date: Mon, 24 Aug 2009 00:15:46 +1000
Mime-Version: 1.0
Content-Type: text/plain
Content-Transfer-Encoding: 7bit

This is plain text
------=_mimepart_4a914f0c911be_6f0f1ab8026659
Content-Type: text/html; charset=UTF-8
Content-ID: <4a914f12cf86_6f0f1ab802692c@baci.local.mail>
Date: Mon, 24 Aug 2009 00:15:46 +1000
Mime-Version: 1.0
Content-Transfer-Encoding: 7bit

<h1>This is HTML</h1>
------=_mimepart_4a914f0c911be_6f0f1ab8026659--
```

Mail inserts the content transfer encoding, the mime version, the content-id's and handles the content-type and boundary.

Mail assumes that if your text in the body is only us-ascii, that your transfer encoding is 7bit and it is text/plain. You can override this by explicitly declaring it.

Making Multipart/Alternate, without a block

You don't have to use a block with the text and html part included, you can just do it declaratively. However, you need to add Mail::Parts to an email, not Mail::Messages.

```ruby
mail = Mail.new do
  to      'nicolas@test.lindsaar.net.au'
  from    'Mikel Lindsaar <mikel@test.lindsaar.net.au>'
  subject 'First multipart email sent with Mail'
end

text_part = Mail::Part.new do
  body 'This is plain text'
end

html_part = Mail::Part.new do
  content_type 'text/html; charset=UTF-8'
  body '<h1>This is HTML</h1>'
end

mail.text_part = text_part
mail.html_part = html_part
```

Results in the same email as done using the block form

Getting error reports from an email:

```ruby
@mail = Mail.read('/path/to/bounce_message.eml')

@mail.bounced?          #=> true
@mail.final_recipient   #=> rfc822;mikel@dont.exist.com
@mail.action            #=> failed
@mail.error_status      #=> 5.5.0
@mail.diagnostic_code   #=> smtp;550 Requested action not taken: mailbox unavailable
@mail.retryable?        #=> false
```

Attaching and Detaching Files

You can just read the file off an absolute path, Mail will try to guess the mime_type and will encode the file in Base64 for you.

```ruby
@mail = Mail.new
@mail.add_file("/path/to/file.jpg")
```

```
@mail.parts.first.attachment?   #=> true
@mail.parts.first.content_transfer_encoding.to_s    #=> 'base64'
@mail.attachments.first.mime_type   #=> 'image/jpg'
@mail.attachments.first.filename   #=> 'file.jpg'
@mail.attachments.first.decoded   == File.read('/path/to/file.jpg') #=> true
```

Or You can pass in file_data and give it a filename, again, mail will try and guess the mime_type for you.

```
@mail = Mail.new
@mail.attachments['myfile.pdf'] = File.read('path/to/myfile.pdf')
@mail.parts.first.attachment?   #=> true
@mail.attachments.first.mime_type   #=> 'application/pdf'
@mail.attachments.first.decoded   == File.read('path/to/myfile.pdf') #=> true
```

You can also override the guessed MIME media type if you really know better than mail (this should be rarely needed)

```
@mail = Mail.new
file_data = File.read('path/to/myfile.pdf')
@mail.attachments['myfile.pdf'] = { :mime_type => 'application/x-pdf',
                                    :content => File.read('path/to/myfile.pdf') }
@mail.parts.first.mime_type   #=> 'application/x-pdf'
```

Of course... Mail will round trip an attachment as well

```
@mail = Mail.new do
  to      'nicolas@test.lindsaar.net.au'
  from    'Mikel Lindsaar <mikel@test.lindsaar.net.au>'
  subject 'First multipart email sent with Mail'

  text_part do
    body 'Here is the attachment you wanted'
  end

  html_part do
    content_type 'text/html; charset=UTF-8'
    body '<h1>Funky Title</h1><p>Here is the attachment you wanted</p>'
  end

  add_file '/path/to/myfile.pdf'
end

@round_tripped_mail = Mail.new(@mail.encoded)

@round_tripped_mail.attachments.length   #=> 1
@round_tripped_mail.attachments.first.filename   #=> 'myfile.pdf'
```

See "Testing and extracting attachments" above for more details.

Using Mail with Testing or Spec'ing Libraries

If mail is part of your system, you'll need a way to test it without actually sending emails, the TestMailer can do this for you.

```
require 'mail'
=> true
Mail.defaults do
  delivery_method :test
end
=> #<Mail::Configuration:0x19345a8 @delivery_method=Mail::TestMailer>
Mail::TestMailer.deliveries
=> []
Mail.deliver do
  to 'mikel@me.com'
  from 'you@you.com'
```

```
  subject 'testing'
  body 'hello'
end
=> #<Mail::Message:0x19284ec ...
Mail::TestMailer.deliveries.length
=> 1
Mail::TestMailer.deliveries.first
=> #<Mail::Message:0x19284ec ...
Mail::TestMailer.deliveries.clear
=> []
```

There is also a set of RSpec matchers stolen/inspired by Shoulda's ActionMailer matchers (you'll want to se `delivery_method` as above too):

```
Mail.defaults do
  delivery_method :test # in practice you'd do this in spec_helper.rb
end

describe "sending an email" do
  include Mail::Matchers

  before(:each) do
    Mail::TestMailer.deliveries.clear

    Mail.deliver do
      to ['mikel@me.com', 'mike2@me.com']
      from 'you@you.com'
      subject 'testing'
      body 'hello'
    end
  end

  it { should have_sent_email }   # passes if any email at all was sent

  it { should have_sent_email.from('you@you.com') }
  it { should have_sent_email.to('mike1@me.com') }

  # can specify a list of recipients...
  it { should have_sent_email.to(['mike1@me.com', 'mike2@me.com']) }

  # ...or chain recipients together
  it { should have_sent_email.to('mike1@me.com').to('mike2@me.com') }

  it { should have_sent_email.with_subject('testing') }

  it { should have_sent_email.with_body('hello') }

  # Can match subject or body with a regex
  # (or anything that responds_to? :match)

  it { should have_sent_email.matching_subject(/test(ing)?/) }
  it { should have_sent_email.matching_body(/h(a|e)llo/) }

  # Can chain together modifiers
  # Note that apart from recipients, repeating a modifier overwrites old value.

  it { should have_sent_email.from('you@you.com').to('mike1@me.com').matching_body(/hell/)

  # test for attachments

  # ... by specific attachment
  it { should_have_sent_email.with_attachments(my_attachment) }

  # ... or any attachment
  it { should_have_sent_email.with_attachments(any_attachment) }

  # ... by array of attachments
  it { should_have_sent_email.with_attachments([my_attachment1, my_attachment2]) }       #note that order is importan
```

```
  #... by presence
  it { should_have_sent_email.with_any_attachments }

  #... or by absence
  it { should_have_sent_email.with_no_attachments }

end
```

Core Extensions

The mail gem adds several constants and methods to Ruby's core objects (similar to the activesupport gem from the Rails project). For example:

```
NilClass::blank?
NilClass::to_crlf
NilClass::to_lf
Object::blank?
String::to_crlf
String::to_lf
String::blank?
...etc...
```

For all the details, check out lib/mail/core_extensions/.

Excerpts from TREC Spam Corpus 2005

The spec fixture files in spec/fixtures/emails/from_trec_2005 are from the 2005 TREC Public Spam Corpus. They remain copyrighted under the terms of that project and license agreement. They are used in this project to verify and describe the development of this email parser implementation.

http://plg.uwaterloo.ca/~gvcormac/treccorpus/

They are used as allowed by 'Permitted Uses, Clause 3':

```
"Small excerpts of the information may be displayed to others
 or published in a scientific or technical context, solely for
 the purpose of describing the research and development and
 related issues."

 -- http://plg.uwaterloo.ca/~gvcormac/treccorpus/
```

License

(The MIT License)

Copyright (c) 2009-2016 Mikel Lindsaar

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

© 2016 GitHub, Inc. Terms Privacy Security Status Help

Contact GitHub API Training Shop Blog About